Ammar Abu Shamleh (21521274)
Pradyumn Vij (21469477)

Parallelisation of column-wise matrix collisions using OMP and MPI

HIGH PERFORMANCE COMPUTING – PROJECT 2

1.0 Problem

A large set of floating point numbers reside in data that is ordered in rows and columns, described as lines in text files.

Each column must have blocks of 4 numbers extracted. A quartet of numbers are termed a block if they are all within the same arbitrary range of each other. The arbitrary range will be referred to as the "DIA". For example, in the trivial case where a column consists of {0.0, 0.1, 0.25, 0.5, 0.6}, the blocks present when the DIA of 0.5 are {0.0, 0.1, 0.25, 0.5} and {0.1, 0.25, 0.5, 0.6}.

Each row has been provided with a unique large integer, and based on project statements, a Block's signature is generated by adding each the four element's row keys together. An assumption that this signature is unique for each block (within the column in which it's found) is based on the project documentation provided.

The task is to find all collisions in the data set. A collision is defined by blocks with the same signature in different columns.

1.1 Problem Statement

Implement the task described in section 1.0 by using the multithreaded OpenMP (OMP) code from Project 1 and running it on the cluster of nodes using MPI.

1.2 Problem Approach and Hypothesis

A considerably well multithreaded program was written using OMP calls in Project 1 to solve the problem and have it run in optimal time. The main issue is the conversion of an OMP program to run across several nodes while also improving the speed of the program. The primary bottleneck in MPI programming is that for any efficiency, the

round-trip time to send and deliver information between nodes and reassemble the data to find the result must be less than the computation time on each node.

Our hypothesis is that the single node computation will take far less time than sending, delivering and reassembling the results of the node's computation.

2.0 Sequential program design requisites

2.1 File input data parsing and data structure designs

The keys were supplied in a file named "keys.txt". The floating point data (i.e the data matrix) was provided in a file named "data.txt". The first task faced in approaching this project was to read all data residing in these two files into internal data structures for use in the program. While this could be easily accomplished using static arrays, the task was instead approached using dynamically allocated arrays. The advantage of this approach is that it doesn't tie the code to the specific data set that has been provided, and makes the code more robust for dealing with other data set sizes.

2.1.1 Data Input - Sequential (O(n))

The floating point numbers provided in data.txt were not prescribed to any specific decimal place, while the example provided shows numbers to 6 decimal places, no restriction was provided. To ensure robustness, the data type of double was used for the extra precision, in the case that more precise results are required from future data sets.

The keys were chosen to be of type long long. The reason was to avoid the large integers from overflowing on any 32 bit or lower architectures. The numbers themselves, within the provided data set, are fine as integers but when added to create a signature, are at risk of causing overflow errors when calculating signatures.

In the following sections the variable "n" refers to the number of inputs values per text file.

2.1.1.1 readMatrix

The function "readMatrix" counts the number of rows and columns of a matrix while dynamically allocating memory and assigning values. A new row is defined by a new line in the text file and a new column for each comma separated number in a line.

2.1.1.2 readKeys

The function readKeys reads the keys.txt file into an array of long long of size ROWS defined in function "readMatrix" (see section 2.2). The function readKeys runs in time and space complexity O(n).

2.2 Data Structures

2.2.1 Block

The Block struct holds the signature of a unique block of 4 elements from any one column in type long long. The entire program hinges on the assumption that signatures are unique for every unique quartet of elements found.

The column from which the block originates is also stored.

In addition, to aid with the post-processing merging of blocks, each block's unique combination of 4 row-numbers are stored as an array of integers.

2.2.2 Collision

The Collision struct is required to accumulate all possible collisions prior to post processing.

An array of integers named "columns" holds the columns in which the collision's blocks were found. For example, if 3 blocks with the same signature are found across columns 4, 6 and 15, then the array will hold these 3 column numbers. The integer value "numBlocksInCollision" holds the number of values in the array (also equal to the number of blocks that were found as part of the collision).

2.2.3 Pair

The Pair struct is required by the optimised sequential block generation code to hold the signature value for the row key value, in the form long long and an integer value describing the original row index. The requirement of this struct is to remember these values after sorting which is required by the approach developed for this algorithm. This struct is once again used in the parallelisation of this code.

3.0 Memory management for parallelisation implementation

Parallelisation often brings average or passable mistakes in sequential coding into the spotlight. Memory management is quite important in all languages and it is important to realise when to optimise how memory is managed.

A 2d array is used to hold the data in long format however the array is pointing directly to the memory for the structs, not to pointers to the structs. The latter causes very poor performance as the pointers are unlikely to all be pointing to continuous data. A pointer also further slows any operation to that specific portion of memory.

Matrix transposition was performed on the main data set as columns of floating point numbers needed to be operated upon (non-contiguous) rather than rows (contiguous data). Transposing the matrix of data improved performance in general as well as for multithreading as it is far more cache friendly (likely to produce a greater number of cache hits rather than cache misses).

3.1 Managing Access to shared Data during parallelisation

One issue that needed to be considered was the access to shared data structures in multi-threaded code. Having multiple threads carelessly access shared data-structures, and attempt to write to shared data-structures, simultaneously, can introduce several issues. In general, managing access by multiple threads to shared data was an issue that required careful examination and thought.

There were two issues for consideration here:

1. Ensuring that thread's simultaneous access to shared read-only data doesn't introduce too many cache misses
2. Ensuring that threads can simultaneously write blocks and collisions they find without memory clashes

3.1.1 Managing access to shared read-only data

As threads work on shared data, it is important that the data is split carefully among the working threads, to maximize cache-hits. If an array has 4000 rows (as in this case, with the data set), and the data is split carelessly into blocks 0-1000, 1000-2000, 2000-3000 and 3000-4000, then such parallelization will incur many cache misses.

A more effective method of splitting the data was required, to temporal and spatial locality within the code. A better approach found was to have the data split using thread numbers and number of threads instead.

3.1.2 Allowing multiple threads to write to a block database simultaneously

As blocks are found, they are to be written to a block database (i.e. an array of block structs). When multi-threading, there will be multiple threads running simultaneously, and having multiple threads write to a shared data space in parallel could introduce severe issues. As such, an alternative approach was found:

Each thread will maintain its own private database of blocks, and will add any blocks it finds to this private database. When a thread has finished its processing work, the private database it created will be merged with a collective, shared database, which is the final storage location for any blocks found. This merging step takes place inside a critical region of code, so only one thread can have write access to the shared database at any one time. While the critical region does introduce overheads that may damage performance, the parallelization still managed to attain excellent speedups. An identical approach was used for the storage of collisions.

4.0 OMP parallelisation

A function (findBlocks) was written for the 2d array of floating point number data (dataMatrix) to iterate over each column and find all Blocks (within each column) with any given DIA. When a block is found it is stored in an array of Blocks.

4.1 Identifying Blocks in data

4.1.1 Sequential brute force - $O(kn^4)$

The starting approach to the project was a simple brute-force, naïve algorithm to find blocks. The algorithm simply contains a loop over all columns, with each iteration making 4 nested loops over all rows to find all blocks. The array of blocks is dynamically reallocated to fit each new block that is found (preventing memory wastage by statically allocating too large an array, or a segmentation faults by allocating too small an array). The approach was akin to:

Figure 1: Finding all permutations of length 4

```
//A loop over all columns
for(int col=0; col<MAX_COLS; col++) {
  for(int r1=0; r1,MAX_ROWS; r1++) {
    for(int r2=r1+1; r2<MAX_ROWS; r2++) {
      for(int r3=r2+1; r3<MAX_ROWS; r3++) {
        for(int r4=r3+1; r4<MAX_ROWS; r4++) {
          //Check if all 4 elements are within DIA
          //If so, generate and store a new block
        }
      }
    }
  }
}
```

As can be seen above, the work done in each column is independent. Additionally, the outermost loop on rows (the loop on **r1**) also has iterations that are all independent of one another. Each loop of that outermost loop on rows will find an independent, unique set of blocks. So both the loop on columns, and the loop on **r1**, were identified as potential targets for parallelization. Another note is that because of the way the loop works, with each subsequent index beginning at the value above the previous index (i.e. **r2** starting at **r1+1**), it is guaranteed that no duplicate blocks will be found.

The time complexity of this approach is O(n^4) per column, where n is the number of rows. Thus, the overall time complexity is O(kn^4), where k is the number of columns, and n is the number of rows. One limitation with this approach is the fact that it only works for block sizes of 4. A change in the block size would entail changing the code to adapt to the new block size (not a desirable feature for robust programs, where such attributes should be freely modifiable).

4.1.1.1 Results

Using a processor timer for timing execution was deemed inappropriate, as it does not give a fair description of the task time taken, especially when timing multi-threaded code. An elapsed time approach was required, using the omp.h library's omp_get_wtime function.

With all **500 columns** and a **DIA of 10^{-6}**, this algorithm takes **2 minutes and 43 seconds** to run. When running only the first **499 columns (same DIA)**, the algorithm takes **26 seconds** to complete on the test machine. All **500 columns** were found to have **29,321,366 blocks**, while the first **499 columns** only have **14,863 blocks**. It can be seen here that the last column (the 500th column) contains the clear majority of blocks in the data.

4.1.2 Multithreaded brute force – O(kn^4), Parallelise by Column

The initial approach to parallelization of the block generation function was to split the work-load by column, i.e. have each thread work independently on a different column. This was considered a viable approach, as the work done in each column is independent of the work done in all other columns. Accomplishing multi-threading in this

manner was simple, as the outermost loop iterates over columns (as seen in the earlier image). Thus, it was a simple case of parallelizing that outermost loop using omp's provided pragmas. While parallelization was achieved, there were bottlenecks that prevented the approach from achieving a good speed-up factor.

Primarily, a single column was discovered to hold most blocks in the test data set provided.

4.1.2.1 Limitations

As can be seen in **3.1.1 Results**, column 500 holds a substantial number of blocks. As such, splitting the work load by column was clearly limited by a single thread having to work on a column that is heavily populated in the provided data set. This load-balancing issue resulted in a poor speed-up factor. We felt this accurately portrayed the types of issues that are encountered in parallelizing the processing of real-world data: perfect load-balancing is difficult/impossible to achieve, in many cases.

If the blocks were evenly distributed across columns, this approach would have worked far better.

4.1.2.2 Results

Unfortunately, due to most of the processing work taking place in the last column, which is still only processed by a single thread, no measurable speed-up was attained by this approach.

4.1.3 Multithreaded brute force – $O(kn^4)$, Parallelise within Columns

Given that load-balancing issues were faced with the previous approach, a new, revised approach was considered. Instead of splitting work by columns, the work would instead be split within columns. This would allow for the effective parallelization of data with very uneven distribution of blocks among columns.

So, to address the load imbalance discovered in the previous attempt, parallelization would occur within each column, rather than across the columns. So rather than assigning columns to threads, the outermost loop of the work done **within** each column was distributed evenly among the threads. Thus, the parallel region begins and ends within each column.

The column's elements were divided by the number of threads, and each thread would individually find a subset of blocks within a column (updating a thread-private database). These partial sets of blocks would then be merged, within a critical region (to avoid race conditions and simultaneous memory accesses by multiple threads to the same region in memory), with a collective, shared database of blocks. Thus, the workload within each column would be split among the multiple active threads.

The work allocation was done using **omp_get_thread_num()** and **omp_get_num_threads()**. Each thread would begin work on an index (over the outermost loop over rows) equal to its thread ID number, and this index would be incremented by the number of active threads. Thus, if 4 threads are active, thread 0 begins work at index 0, and increments its index by 4 on each subsequent iteration. This

approach is much more cache-friendly then splitting the loop into contiguous chunks for each thread to work on (i.e. assigning thread 0 iterations 0-1000, thread 1 iterations 1000-2000, thread 2 iterations 2000-3000 etc.). The reason is that with this approach, the threads all remain within roughly the same region of the data as they process it, and thus referential locality of the code is maximized, and so too is the cache-hit to cache-miss ratio.

The method looked something like:

Figure 2: Parallelised method to find all possible computations of length 4

```
//A loop over all columns
for(int col=0; col<MAX_COLS; col++) {
  #pragma omp parallel
  {
  //BEGIN PARALLEL REGION
          Block *privateDatabase = malloc(......);
          int ID = omp_get_thread_num();
          int nthreads = omp_get_num_threads();
          for(int r1=ID; r1,MAX_ROWS; r1 += nthreads) {
            for(int r2=r1+1; r2<MAX_ROWS; r2++) {
              for(int r3=r2+1; r3<MAX_ROWS; r3++) {
                for(int r4=r3+1; r4<MAX_ROWS; r4++) {
                  //Check if all 4 elements are within DIA
                  //If so, generate and store a new block
                }
              }
            }
          }
  //END PARALLEL REGION
  }
}
```

This approach to parallelisation was investigated to try to introduce multi-threading on the heavily populated 500th column, which was found in prior approaches.

One downside to the approach is that with parallel regions beginning and ending per column, the overhead of creating new threads, and the overhead of encountering the critical region, are now encountered multiple times (once per column for generation of new threads, and once per thread per column for critical region). Thus, while more even load-balancing is attained, it comes at the cost of increased overheads. For columns with very, very few blocks, this could result in worst performance with the parallel code (as there is not enough computation work being done to justify the overheads involved in the parallelization).

4.1.3.1 Special Case Observations

While the basic approach described above worked extremely well for the first 499 rows, it didn't attain any speed-up (and in fact caused slowdown) for the last column.

This was addressed by writing a special case for the last column, where by the last column would be split among a set of threads using tasks. A single thread would generate tasks, each of which perform processing on a certain chunk of the last column. These tasks are added to a task pool that is worked on by any available threads. This

improved performance for the last column compared to the simple split of the for-loop work, but there was still slight slowdown compared to the sequential code.

On further examination, it appeared that the load balancing problems that were occurring at a column level were also occurring at a section level, as there were far more blocks in the first and second sections of the column than the rest, leaving idling threads. Thus, the size of chunks generated by tasks had to be revised, to more evenly distribute work among threads and maximize core utilization.

This problem was targeted by reducing the chunk size of each task, and thus providing more threads for the first 2000 elements. While this revision improved load balancing, it reduced performance for some reason. It seems that when multiple threads were running simultaneously on the last column, they would individually each work slower than when fewer threads were running.

These observations strongly suggested memory access problems; that each thread's execution was interfering (at a cache level) with the other active threads. After multiple failed attempts to rectify the issue, the last column was simply discarded, and the program would hence only run on the first 499 columns.

4.1.3.2 Special case limitations

The most obvious limitation is that this approach (with the tasking on the last column) would only work on the provided data set.

4.1.3.3 Special case results

Memory management issues were identified. Load-balancing is a difficult issue to completely resolve for unpredictable, real-world data

4.1.3.4 Introducing transposition

To address the memory mismanagement issue described earlier (with the outermost loop being over columns), the data matrix was transposed so that the columns are stored in rows, thus reducing cache-miss frequency substantially.

Matrix transposition only costs O(n) in time complexity.

4.1.3.5 Limitations

One limitation is the issue described earlier, concerning the parallel region beginning and ending once per column. This incurs lots of overheads.

A second issue is that each thread requires its own private database of blocks (to avoid concurrency issues by writing to a shared database). A merge of these databases would be required at the end of each thread's work in a column. This merging would take place in an omp critical region, which is damaging to performance. The critical region would be encountered k * t times, where k is the number of columns and t is the number of threads, an example would be 4 threads with 500 columns resulting in 2000 encounters.

4.1.3.6 Results

The result on the first **499 columns** (excluding the populous column), **before** matrix transposition is a multithreaded run time of block-generation that takes an average of

12.406089 seconds versus an average of approximately **26 seconds** for the base sequential code.

This is a speed-up of a factor of **2**.

After transposing the data matrix, the sequential run-time dropped down to about **19.172642 seconds** (from **26 seconds**), and the parallel run-time dropped down to about **4.854188 seconds**, achieving a speed-up factor of **4** with the parallel code. Thus the cache-miss resolution increased the parallel speed-up factor significantly (roughly doubled it), reinforcing the idea that memory management is crucial for effective parallelization.

4.1.3 Sequential Sorted Combinatorics – $O(kn^4)$

An algorithmic improvement of the brute force would clearly improve the run time of finding the blocks. This method uses the dataMatrix's columns (now rows after transposition).

For each column, a temporary array of Pairs is created. Where a Pair consists of a value, which is the actual value in the data Matrix (a double), and a key from the key array (a long long). This temporary array of pairs is sorted according the value (double) of the pairs.

The next step is to have a starting pointer and an end pointer defining a region of size 4 or more where the difference between the upper bound minus the value of the lower bound is within DIA. All the values between the upper and lower bounds can construct valid unique combinations of size 4 to create blocks. Thus, only unique blocks are ever calculated.

Figure 3: Pseudo code for optimised block generation, the method only computes valid unique bocks and adds them to the database

```
Block *sequentialFastBlockGen() {
    pair* pairContainer;
    for( each col in COLS of the Database of dataMatrix) {
        for( each row in ROWS){
            add an entry to pairContainer for value, key and row index
        }
        lower = 0
        sort(pairContainer by entry.values)
        for(each entry set it as an upper bound){
            while(the difference between upper and lower is greater than DIA){
                ++lower bound
            }
            find all unique combinations of size 3 inclusive of lower, exclusive of upper
            place the combination + 1 inside of container of blocks
        }
    }
    free(pairContainer);
    return blockDB;
}
```

The time complexity of sorting is O(nlgn).

The cost of iterating across the column is O(n).

The cost of finding all unique combinations in a window is $O(w^3)$, where w is the number of elements in the window (greater than 4).

The resultant complexity is $O(k(nlgn + nw^3))$.

In the very worst case, every element in the array is within DIA which makes the time complexity $O(kn^4)$. However the average case is O(kn + B) where B is the number of valid blocks in the data provided.

4.1.3.1 Results

All following results are with a **DIA** of **$2.5 * 10^{-6}$**

The algorithm (sequentially) finds all blocks in the first **499 columns** in **0.354579 seconds**

When trialled over **all columns** (inclusive of the populous **500th column**), an average run time of **0.906457** seconds was achieved during testing.

These are certainly impressive results, and indicate that the worst case is highly unlikely.

Compared to the brute force sequential algorithm, the optimized code (on this data set) for the first **499 columns** ran an average of **54.07 times faster**. In addition, it runs **13.689 times faster** than the best multi-threaded attempt (which took **4.85 seconds**) on the brute-force code.

There were several interesting observations that were derived from this improvement, concerning the efficiency improvements that can be gained by parallelization vs. Algorithmic improvement. Parallelization focusses on using hardware to split existing workload among multiple cores; the instruction count (IC) of the program remains unchanged. Thus, runtime is reduced by using multiple cores for execution. Conversely, algorithmic improvements focus on minimizing the instruction count (IC) of the program through logical redesign. While parallelization speed-up factors are severely limited by the number of cores available, and the overheads involved in creating new threads, radical algorithmic redesign can (as shown in this example) drastically improve performance of a program. This introduces several interesting questions about whether it is better to focus on designing code to be parallelizable, or whether it is instead more efficient to focus on major algorithmic improvements, at the cost of parallelizability. Ultimately, we felt that the answer to this question is severely context dependent, and will change depending on which approach gets better performance for the specific problem being solved.

4.1.4 Multithreaded Sorted Combinatorics – $O(kn^4)$

Given that the optimized code still treats columns independently, the best approach to parallelization was the simplest: simply parallelizing the outer for loop that loops over the columns.

Simply creating an **omp parallel for** pragma around the outermost (loop over column) for-loop yielded an impressive speed up, as each column was being dealt with completely independently, while the computations within the inner for loops are too dependent to easily multithread. The complexity stays the same as above but the

multithreading directly addresses the k portion of the complexity (i.e. the work-load of k columns is split among multiple cores, reducing run-time).

4.1.4.1 Results

Results below are with a **DIA of 2.5 * 10^{-6}**

For the first **499 columns,** the algorithm takes an average of **0.097343 seconds** to execute. Compared to the sequential sorted combinatorics algorithm, this provides a speed-up factor of **3.64257**.

For all **500 columns,** the algorithm takes an average of **1.707430 seconds** to execute. This introduces slowdown compared to the sequential code. It runs roughly **1.9 times** slower than the sequential optimised code for all **500 columns**.

The inclusion of the load-unbalancing 500th column is a clear example of where a multithreaded approach can be detrimental based on the data set. For situations when the load is heavily favoured to one column (such as this data set), there is a reduction in speed by a factor of 2. However, in cases where the balance is spread over multiple columns, the increase in speed can be significant, as shown above, where it's sped up close to **3.5 times.**

4.2 Identifying Collisions amongst Blocks

After finding and storing all Blocks in a block database, the blocks involved in collisions must be identified, within the function findCollisions. This function's purpose is to determine which signatures exist in more than one column, and then determine the set of columns over which each signature involved in a collision occurs.

4.2.1 Sequential Brute Force – O(n^2)

This naïve method involves creating a Boolean array of the same numerical size as the array of Blocks. The function is to check each Block's signature and check it against all other blocks in the array of blocks. If other blocks with matching signatures are found, the number of occurrences (i.e. the number of blocks with the same signature) is recorded and stored in a Collision, along with the set of column numbers in which the same signature was found. This collision is then stored in an array of Collisions. In the process, any blocks found in collisions are marked as true in the relative position within the Boolean array, to avoid repetitions. i.e if blocks 1 ,5 and 7 are found in a collision, then positions 5 and 7 in the Boolean array are marked as true, to avoid blocks 5 and 7 later being found in a duplicate collision.

The time complexity of this function is O(n^2), where n is the number of blocks. The space complexity required is O(B + b + c), where B is the number of Blocks; b is the number of Booleans (equal to the number of Blocks, but smaller as storing a Boolean rather than a Block); c is the number of collisions.

4.2.1.1 Results

Results below are with a **DIA of 2.5 * 10^{-6}**

For the first **499 columns**, to find all collisions the algorithm takes an average of **71.75 seconds** to finish.

For all **500 columns,** the algorithm failed to finish.

4.2.2 Multithreaded Brute Force – O(n^2)

The method described in section 3.3 is similar to the approach taken to optimise the performance of brute force in this particular situation.

For the first 499 columns, the algorithm takes an average of **20.374063 seconds** to finish. A speed-up factor of **3.52**

With all **500 columns**, the algorithm failed to finish.

4.2.3 Sequential Sorted Blocks – O(nlgn)

Similar to block generation, the next step was an algorithmic redesign on the base sequential code. This time, the algorithmic approach to the collision detection within the database of blocks was substantially improved

The most optimal solution found is to sort the array of Blocks by signature, which costs time and space O(nlgn).

The next step is to iterate through the sorted blocks looking for identical signatures. Now that the array is sorted by signatures, all identical signatures (and thus all collisions) will be grouped up, and easy to find. If more than 1 contiguous identical signature in the array is found, all block's columns in the contiguous signature instance are added to a single collision. This second part of the solution also builds the array of Collisions and costs O(n).

4.2.3.1 Results

Results below are with a **DIA of 2.5 * 10^{-6}**

For the first **499 columns**, to find all collisions the algorithm takes an average of **0.060873 seconds** to execute, providing a speed-up factor of **1179** compared to the sequential brute-force code. Similarly, compared to the parallel brute-force code, the speed-up factor here is **334.6**.

For all **500 columns**, to find all collisions the algorithm takes an average of **9.000617 seconds** to execute.

This again illustrates how algorithmic redesign of code can achieve substantially higher speed-ups than parallelization alone can ever provide.

4.2.4 Multithreaded Sorted Blocks – O(nlgn)

Parallelizing the optimised collision detection algorithm proved a more involved task than all earlier attempts at parallelization. The reason is that there were more complex dependencies within the for-loop. In earlier stages, the outermost loop's iterations were always independent of one another, allowing for simple parallelization.

However, the issue faced with this new algorithm was that there was only one for-loop over the array of blocks (now sorted by signatures, thus collecting collisions together within the array). The problem here is simple parallelization of the for-loop would destroy the correctness of the program, as the array was effectively split into variable sized chunks, with each chunk (a chunk being a segment of the array where the signatures of the blocks are all equal) represents a collision. Thus, each chunk can only be worked on

by one thread at a time; given that the chunks are variable sized, and could begin and end anywhere in the array, this made parallelization a complex task.

Consider this example:

Figure 4: Example set of blocks with signatures

```
Block Array
Block 1: Signature 11201
Block 2: Signature 11201
Block 3: Signature 11201
Block 4: Signature 11201
Block 5: Signature 11402
Block 6: Signature 11504
Block 7: Signature 11504
Block 8: Signature 11705
Block 9: Signature 19120
......
......
......
```

This effectively represents the way the algorithm proceeded. The issue here is that the loop over these blocks can't be parallelized by a simple **omp for pragma**, as a collision might be cut in half, with one half allocated to one thread, and the other half allocated to another thread. This would then result in an incorrect result. For instance, in the above example, careless parallelization of the for-loop could split the first collision (on signature 11201) among 2 threads, thus hampering correctness of the program.

The solution found was to use tasking to split the work up. A single thread splits the array into multiple roughly evenly sized chunks, and ensures that each chunk begins and ends on a 'collision boundary' i.e. no collision is chopped up between two chunks. The single thread then generates tasks for the collision detection within each of these chunks, and these tasks can then be worked on independently and simultaneously by multiple threads, which will each develop their own partial database of collisions over the chunk that they have been assigned.

These partial databases are then merged into a single complete database inside a critical region (similar to how the partial block databases are merged).

4.2.4.1 Results

Results computed with a **DIA of 2.5 * 10^{-6}**

Even with the added overhead of the critical regions, a significant speed-up factor was still achieved.

For the first **499 columns,** the algorithm takes an average of **0.031493 seconds** to execute. This achieves a speed-up factor of **1.93**.

For all **500 columns,** the algorithm takes an average of **3.411191 seconds** to execute, providing a speed-up of **2.64.**

One observation made here is that the speed-up factor attained by parallelizing the optimised collision detection is not as high as the speed-up factor attained by parallelizing the brute-force collision detection. The speed-up factor attained (on **499 columns**) through parallelization of the algorithmically optimized code was roughly **1.93**, in this example. Conversely, the speed-up factor attained through parallelization on the brute-force collision detection code was roughly **3.52**. This is a result of the algorithmically optimized code being less parallelizable, due to the complex dependencies that are introduced into the for-loop by the new approach. However, given that the algorithmic optimization provides such a substantial speed-up factor (through a radical reduction to the program's instruction count, by reducing complexity from $O(n^2)$ to $O(n\log n)$), we believe that sacrificing a bit of parallelizability for the incredible improvement to performance is a very worthy trade-off.

A potential reason why the speed-up factor here is not as high as with the brute-force collision detection could be a result of the inherent cache-unfriendliness of the way the data is being split. The block database is split into roughly equal-sized chunks, which are allocated to threads. The issue is that (due to the nature and complexity of the sequential algorithm), these chunks are split in a manner that increases the frequency of cache-misses during execution. i.e. one task may operate on a chunk from 0-2002, another task from 2002-3007, another from 3007-4009 etc. This manner of splitting work was previously described (within this report) as being cache unfriendly, due to not exploiting spatial locality. Despite this issue, there is no other effective way to split the collision detection up among multiple threads. Consequently, the multi-threaded performance suffers, and the speed-up factor is diminished.

This example (along with the similar algorithmic optimization of the block generation) demonstrates the inherent bottleneck in parallelization compared to algorithmic redesign: parallelization speed-ups are severely limited by the physical number of cores in the machine, which, on average modern machines, is only 4. In addition, parallelization is also critically dependent on careful memory management, and performance gains can be damaged by too much synchronization or too many overheads in parallel code. Thus, if radical algorithmic improvements can be made to code, this is often a better starting point for performance improvements than parallelisation alone.

5.0 MPI parallelisation

5.1 Block-Generation

The block-generation used for MPI was the best performing block-generation algorithm for OMP, which was the sequential sorted combinatorics algorithm (*Section 4.1.4)*.

Referring to *4.1.4*, the block-generation has dependencies only across columns, and no dependencies within them. This dependency, however, had already been targeted by OMP multi-threading, which parallelised across the columns using different cores on a workstation. Thus, to further parallelize using MPI, the columns were first split among the active nodes (by MPI), and then further split (within each node) among the cores (by OMP).

So, for instance, given 4 nodes and 500 columns: 1 node would be the master node, thus leaving only 3 nodes to perform computation. The 500 columns would be split among the 3 nodes, giving each node roughly 166 columns. On each node, OMP would then further split these 166 columns among the active cores (using the same multi-threaded algorithm described in *4.1.4)*.

Each node will, knowing its rank, the communicator size, and having access to the data files, calculate the chunk of the data that it should perform computation over. The node will, after performing this computation (using the same algorithm from *4.1.4*, over the data set it has been allocated within the cluster), will then send the blocks it finds to the master node, through MPI_Send() commands. Rather than sending block structs, we elected to send (as basic MPI datatypes), the values that constitute a block struct, and then rebuild the struct on the master node's end. The master node maintained (as it received blocks from slave nodes) a collective database of all the blocks found.

5.1.1 Results

Compared to the OMP block-generation (with 4 threads), our MPI block-generation took (at the best runtime, achieved over 18 nodes, 4 threads per node), **0.414910 seconds.** Conversely, the OMP block-generation took **0.098275 seconds.** As can be seen, the combined MPI/OMP algorithm runs significantly slower then the OMP algorith, with a slowdown factor of **4.22.**

This is an expected and justifiable result. It stems from the excessive communication that must be performed to parallelize the algorithm across MPI, as each block found by the slave processes must be communicated back to master. In addition, while the slave processes are computing their partial block databases, the master process performs no work, simply idling, which is an ineffective use of the available resources. However, the predominant cause of the slowdown is the aforementioned communication of all blocks from the slave nodes to the master node. This is an extremely large amount of communication, and would only be justified were there an even larger of computation being performed by each node. In this case, however, with the size and scale of the problem, and the amount of communication it requires, the communication costs outweigh the distribution of the computational work, thus resulting in a significant slowdown.

5.2 Collision Detection

The parallelization of the collision detection using MPI is very similar to the parallelisation of the block-generation using MPI. The block database, over which collisions are to be detected, is first sorted by master (as this is a requirement of the algorithm, described in *section 4.2.4*), then split among the nodes. Each node then further splits its partial database among its cores (using algorithm described in *4.2.4),* and thus finds all the collisions in its partial block database. These collisions are then communicated back to the master node, which adds them to a collective database of collisions.

The main difference between this algorithm, and that described in *5.1,* is the fact that for this algorithm, the master node first has to communicate each slave's partial block database to it, whereas in *5.1's* algorithm, the slave nodes could immediately begin their work without requiring any receipt of data from the master node.

The reason for this difference is that the slave nodes in *5.1* operated on data they could obtain directly from the data files, and thus did not need to receive any data from the master node. In this case, they must operate (as described in *section 4.2.4*) on a sorted block database. Only the master node has this collective sorted block database, and thus must communicate each slave node's portion to it before they can begin computation. Unfortunately, the slave nodes cannot operate on the partial block databases they previously computed (in *5.1*), as these must be merged with master, and the entire collective database sorted, to ensure correctness of the program. As such, the communication step between master and slave nodes is necessary, and cannot be worked around.

5.2.1 Results

As is to be expected, the combined MPI/OMP algorithm performs exceedingly poorly. The best runtime achieved is a runtime of **0.746108 seconds.** By comparison, the OMP algorithm achieved a runtime of **0.031493 seconds.** Thus the OMP/MPI implementation achieved a slowdown factor of **23.7** compared to the OMP implementation.

There are 2 primary for this slowdown.

The first is the communication cost, which is much higher for this algorithm than it was for the MPI/OMP algorithm for block-generation, as there is much more data being sent (master must send each slave the blocks it has to work with, and then receive from each slave all collisions that slave found).

The second is the fact that, referring to *4.2.4,* the dominating portion of the algorithm is the *O(nlg(n))* sorting step, which in this case, is only performed by master. The second step that is parallelised is only a linear step, which master could compute faster itself, rather then linearly sending each block to slave nodes to perform (as the MPI algorithm described above has done). So on top of the dominating step having to be performed sequentially, the following sequential linear step has actually been replaced with a much more expensive process involving expensive two-way communicational overheads. Thus a severe degradation in performance is expected, and such is the result that was observed.

6.0 Performance Results

Note that all results documented here are **after** matrix transposition and that **dia** used here is **$2.5 * 10^{-5}$**, rather than the default **dia** of **$1 * 10^{-6}$**.

Table 1: Execution times (best times are in bold), all multithreaded tasks are performed on 4 threads, functions that did not finished have DNF values. MPI code was not tested on the cluster for the populous 500th column (termed N/A)

Function	Block Generation (499 columns in seconds)	Collision Identification (499 columns in seconds)	Block Generation (500 columns in seconds)	Collision Identification (500 columns in seconds)
Sequential Brue Force	20.249409	71.753608	163	DNF
OMP Multithreaded	4.649090	20.374063	DNF	DNF

Brute Force				
Sequential Optimised	0.385814	0.060873	**0.906457**	9.000617
OMP Multithreaded Optimised	**0.098275**	**0.031493**	1.707430	**3.411191**
MPI and OMP Multithreaded Optimised (18 nodes)	0.414910	0.746108	N/A	N/A

The primary issue encountered when parallelising OMP programs using MPI, in the search for providing faster execution, is the introduction of latency between nodes. With OMP, extra threads have far less negative effect, as OMP uses shared-memory parallel execution. There are however, overheads present in the way of synchronisation, and when implementing a programming task that requires merging thread private databases by the master thread.

Parallelisation through MPI faced far more difficulties, especially in still attempting to achieve effective speed-up factors, despite the heavy communication costs, and the reliance upon previously computed data for slave nodes. The main bottleneck in parallelising a program using MPI is the communication cost between different nodes. Thus the most effective parallelization would come about by minimizing communication between nodes. Unfortunately, this project was revealed to be poorly suited for a distributed-memory cluster, as the necessity for communication between nodes is excessive and unavoidable. Based on our own testing, and as illustrated in *Table 1* above, MPI consistently achieved no speed-up, and in fact caused significant slowdown, when applied to this problem. The block-generation ran roughly 3 times slower on MPI/OMP then it did on OMP. Similarly, for the collision detection, MPI/OMP ran roughly 15 times slower than the OMP implementation.

One large issue faced in parallelising using MPI was the reliance on previously computed data in the program. When block generation completes, the blocks are all stored in a block database (an array of block structs). The collective database is stored and maintained by the master process. When collision detection comes around, the slaves require the blocks that were computed (in the database) in order to find the collisions. As master process is the only process that maintains a collective, complete database of block structs, these blocks must then be communicated to the slave processes in order for them to find the collisions. This communication bottleneck is extremely damaging to performance, and is reflected clearly in the 15x slowdown factor or collision detection. Additionally, for collision detection, another factor that impedes effective MPI parallelization is the operation of the algorithm. Referring to *Section 4.2.3,* the optimised collision detection algorithm works by sorting the array of blocks by signature, and then iterating through the sorted blocks in linear fashion and marking all collisions found. As can be seen, the dominating factor here is the O(*nlog(n)*) sorting step. Unfortunately, this sorting step is impossible/extremely difficult to parallelize, and is thus performed on the master process, before the linear process is then divided among the nodes available. It is thus the case, here, that the dominating step must be performed sequentially. As such, the MPI parallelization is almost certain to create a slowdown; this is because any performance gains through MPI would be marginal (given that they are on the less time-

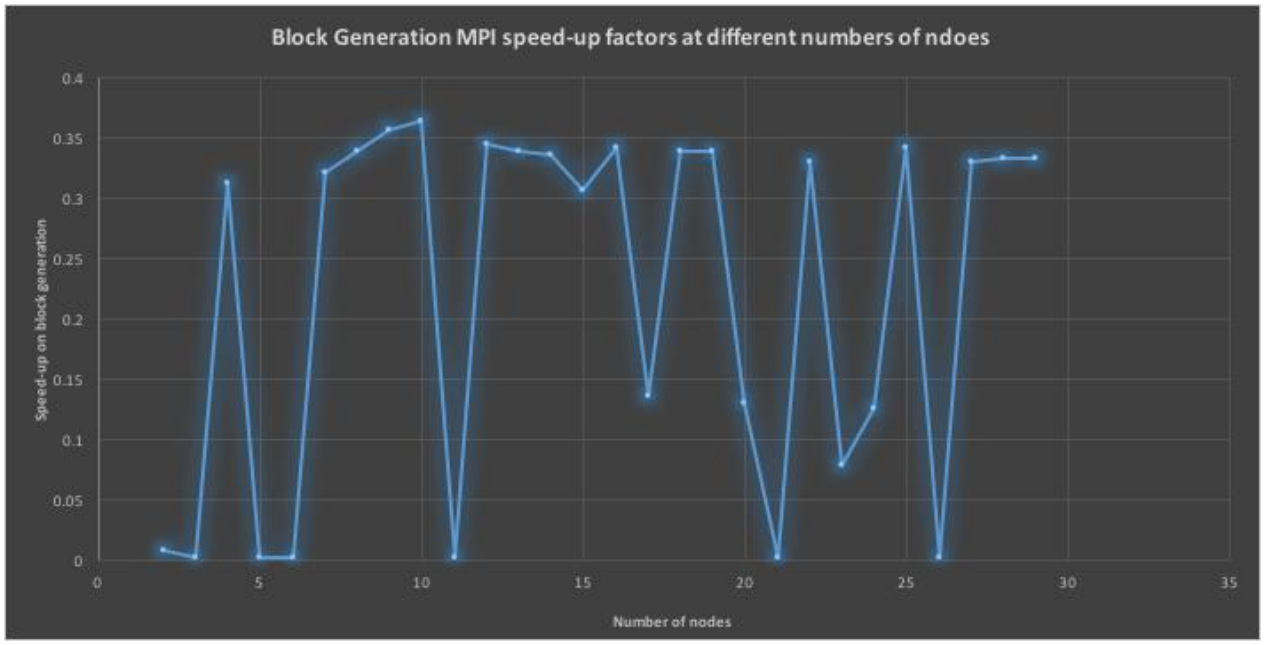
intensive linear step), and easily outweighed by the tremendous communication requirements: the master process must first communicate all relevant blocks to each slave process, and each slave process must (after performing its very minor computational work) communicate all the collisions it finds back to master. These communication costs are not justified by the very miniscule computation that each node performs, and thus performance degrades.

The only case where MPI would be useful for this problem would be if the time taken for each slave to perform its task was higher than time taken to communicate and merge.

The optimised algorithm works ideally on multithreaded systems in which there is shared memory, and no communication costs are incurred.

From our observations, it seems that MPI suits problems that have little to no need for deterministic operations (those which must be performed with specific order, where parallelisation becomes difficult, such as a sort), and problems where the computation performed by each individual node is large enough to justify the communication costs incurred through send and receive operations. In particular, parallelisation becomes increasingly difficult the more dominating deterministic operations (which are often only computable sequentially, on one thread/node) become in the program's execution. This problem, and in particular, our best algorithmic solutions to it, did not satisfy these criteria. As such, the slow-downs were very expected and reasonable outcomes, given the factors and circumstances at work, and the incompatibility of our code with MPI parallelization (and distributed memory architectures in general).

Below are the graphical results of our scalability testing, acquired by running the program across different numbers of nodes, ranging from 2 to 29.

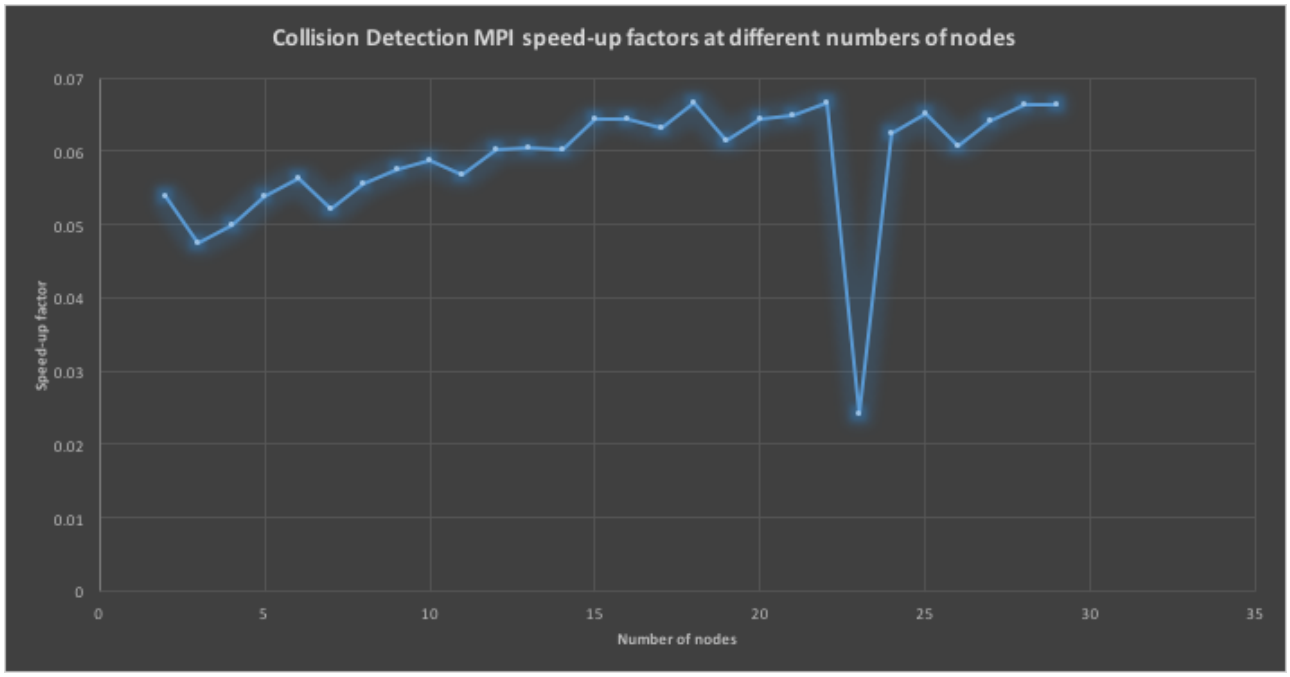


As can be seen above, the results for block-generation across different numbers of nodes are inconsistent. There is no tangible relationship shown between the slowdown factor and the number of nodes, and no conclusions can be drawn, other than that there is a consistent slowdown, in comparison to OMP runtimes.

The speedup factor ranges from 0.00196 to 0.363731. This means the program ranges from being 2.75 times slower to being 510 times slower than an OMP (shared-memory) implementation.

The only conclusion this data provides is that there is a consistent slowdown with MPI. There are potentially multiple other factors that have influenced this data, which have not been considered/discovered. For instance, it was not possible to account for whether or not other users were simultaneously misusing the distributed-memory architecture (the cluster), which may or may not have had bearing on these results. However, the collision detection data below does not suggest that this was the case.

The actual runtimes of the MPI block-generation varied from ~0.4 seconds to ~65 seconds, suggesting an extreme (perhaps external) influencing factor, and suggesting that the graph above does not adequately reflect or illustrate the performance of MPI in general.



The performance of the MPI collision detection is consistently poor (a slow-down factor that fluctuates between 15 and 25, on average). This being said, it is far more consistent than the corresponding graph for the block-generation. The collision-detection slowdown is primarily caused by a combination of 2 factors:

1. The runtime of collision detection is dominated by a sort, which is run sequentially on the full block database by the master process. So any speedup achieved through MPI is only achieved on a smaller portion of the work. Thus the portion of this algorithm that can be parallelised is dwarfed by the portion that can't, making any gains miniscule
2. The communication costs involved in sending all the collision data from slave nodes to the master node heavily outweigh the computation that each node

has to perform. As such, these costs are not justified by the work being performed on each node

Given these factors, especially the 2nd, it is well within the realm of expectation that we would achieve a slowdown by introducing MPI, as the excessive communicational overheads dominate the runtime of the algorithm. It is interesting to note that there is an outlier at 23 nodes, which may be a result of the same unknown influencing factor that distorted the results earlier, as there is no notable trend or reason, based on all the other graphed results, for this sudden decrease.

7.0 Conclusion

From our observations and experiments, it seems that OMP was far more suited to this task than MPI. More generally, this problem is better handled through shared-memory parallelisation than distributed-memory parallelisation. This is a result of the information and data that must be shared by cores/nodes for correct processing and computation of the data.

Our OMP implementations all achieved successful and impressive speedup factors, and most scaled extraordinarily well with the number of cores. Conversely, our combined MPI/OMP implementation performed poorly in comparison. This is predominantly due to the necessity for excessive data communication that exists in the problem, and the lack of sufficient per-node computational work to justify or outweigh this communication.

Parallelisation relies on being able to sufficiently divide and sub-divide the task while still maintaining correctness. There thus must be some level of independence amongst tasks being performed in order to achieve effective speedups. Thus problems with excessive dependencies, such as dynamic programming algorithms, are poorly suited.

As always, the performance of a parallel program running on a cluster depends critically on the performance of its sequential parts. If it is the case that the sequential portion dominates the run-time, then it is highly unlikely that parallelisation will achieve an effective speedup, on either a shared-memory or distributed-memory architecture.

Finally, effective parallelisation across a distributed-memory architecture relies on being able to minimize communication, both in terms of how many times its performed, and in terms of how much data traverses the network. For example, MPI would better-suit a problem with large amounts of computation being performed non-deterministically, in a single data-aggregation scenario (maximal per-node computation, minimal communication). Conversely, a problem requiring excessive amounts of data sharing between individual processing stations are poorly suited to distributed-memory architectures, and better targeted by shared-memory parallelisation.